Cost Effective Physical Register Sharing

Arthur Perais, André Seznec

Cost Effective Physical Register Sharing[1]

Arthur Perais André Seznec
IRISA/INRISA
{arthur.perais,andre.seznec@inria.fr}

ABSTRACT

Sharing a physical register between several instructions is needed to implement several microarchitectural optimizations. However, register sharing requires modifications to the register reclaiming process: Committing a single instruction does not guarantee that the physical register allocated to the previous mapping of its architectural destination register is freeable anymore. Consequently, a form of register reference counting must be implemented.

While such mechanisms (e.g., dependency matrix, per register counters) have been described in the literature, we argue that they either require too much storage, or that they lengthen branch misprediction recovery by requiring sequential rollback. As an alternative, we present the *Inflight Shared Register Buffer* (ISRB), a new structure for register reference counting. The ISRB has low storage overhead and lends itself to checkpoint-based recovery schemes, therefore allowing fast recovery on pipeline flushes.

We illustrate our scheme with *Move Elimination* (short-circuiting moves) and an implementation of *Speculative Memory Bypassing* (short-circuiting store-load pairs) that makes use of a TAGE-like predictor to identify memory dependencies. We show that the whole potential of these two mechanisms can be achieved with a small register tracking structure.

1. INTRODUCTION & BACKGROUND

As most general purpose instructions are destructive in x86 (i.e., one of the source gets overwritten by the result), register-to-register moves are widely used. Those moves consume processor resources and execution bandwidth but would be unnecessary if instructions had distinct destination registers.

In particular, the limited number of architectural registers often forces the compiler to spill variables to memory, only to reload them a few cycles later. Communicating values through memory is slower, since a *load-to-use* delay is introduced.

In both cases, those inefficiencies can be mitigated through *register sharing* at execution time. In the first case, the destination register can simply be mapped to its source register in the frontend, and the instruction potentially removed from the pipeline depending on the particular micro-architecture [1]. Despite constraints due to accesses to partial registers on x86, many instructions can be early executed *non-speculatively* in the frontend (i.e., in-order) in this fashion. Since *Ivy Bridge*, Intel processors implement move elimination (ME) [2].

In the second case, store-load pairs that fit within the processor window can be speculatively identified, and the destination register of the load mapped to the source register of the store [3, 4]. This is referred to as *Speculative Memory Bypassing* (SMB). Although the cache will ultimately have to be accessed to validate speculation, this allows dependents on the load to execute at least *load-to-use* cycles earlier. SMB through the Physical Register File (PRF) has usually been limited to store-load pairs that fit within the instruction window.

Although both optimizations have been extensively covered in previous work [1, 3, 5, 6, 7, 4], studies that implement sharing through the PRF [1, 3, 6] generally failed to precisely describe how register sharing is actually handled. Most studies consider the adjunction of a reference counter to each physical register. The counter is incremented each time the register is assigned as the destination of an instruction, and decremented when an instruction would usually free the register. Mispredictions are handled by sequentially walking through all squashed instructions and decrementing counters accordingly.

We argue that this particular scheme is impractical since 1) The counter array must provision enough logic to increment up to *rename-width* and decrement *commit-width* not necessarily distinct registers on each cycle 2) Recovery latency on a branch misprediction may be increased due to the need to scan squashed instruction and decrement their corresponding counters sequentially. In deeply pipelined architectures, the branch misprediction penalty has been shown to be the largest contributor to performance degradation [8].

Our main contribution consists in an alternative register reference counting scheme that requires low storage. Contrary to reference counters, our scheme supports checkpointing the tracking structure, allowing for a much faster repair of the tracking state on a pipeline flush.

As they are well-known features, we evaluate this scheme on ME and SMB. In the process, we contribute a new implementation of SMB that benefits from a TAGE-like [9] dependency predictor and other optimizations such as load-load bypassing [7]. Using ME and SMB, we find that our sharing framework performs on par with an ideal register tracking

[1]This version of the paper slightly differs from the one accepted at IEEE HPCA'16 (http://dx.doi.org/10.1109/HPCA.2016.7446105). Changes are underlined.

scheme.

In the remainder of this paper, we first describe ME and SMB in more details before introducing the *Inflight Shared Registers Buffer* (ISRB), a small checkpointable structure that enables sharing a subset of the physical register file. We then study how ISRB parameters impact SMB and ME and show that only 480 bits of ISRB storage (plus 96 bits per checkpoint) are sufficient to support both SMB and ME at the same time, on our processor configuration.

2. MOVE ELIMINATION

One of the most intuitive optimization leveraging register sharing is *move elimination* (ME). It was first proposed by Jourdan et al. [1] to address the large number of register-to-register moves in x86 code, since x86 instructions (including x86_64) overwrite one of their source operands. By mapping the destination of the move to the physical register corresponding to its source architectural register, the move can be executed at Rename without consuming a functional unit or an entry in the scheduler.

2.1 Move Elimination for x86_64

Although ME appears as very intuitive, the ability to reference partial registers in x86 renders a practical implementation quite complex. First, moves from 16- and 8-bit registers cannot be eliminated.[2] Similarly, moves with zero extension from an 8-bit register cannot be eliminated if the register is the high part of a 16-bit register (e.g., AH for AX). Those rules as summarized in Intel's Optimization Guide [2].

As a result, even without the issue of register reference counting, ME is not straightforward for x86_64. Yet, it has been implemented in Intel processors since Ivy Bridge [2] and AMD processors since Phenom [10].

2.2 Reference Counting

In [1], reference counting is done by associating one counter to each physical register. How counters are rolled back on a pipeline squash is not covered in the paper, hence we assume that instructions on the wrong path are processed sequentially (potentially by chunks) and their associated counter is decremented.

Similarly, Intel does not disclose how reference counting is implemented. However, Intel's documentation [11] and a patent [12] hint that not all physical registers are tracked by the reference counting mechanism. Indeed, among the performance events that are available on current generation processors, one relates to **unsuccessful** move eliminations [11]. In particular, it tracks the *"Number of integer Move Elimination candidate uops that were not eliminated"*. A similar event exists for SIMD moves.

To our knowledge, the only reason for which a move would not be eliminated would be if the number of registers shared at any given time were limited. This assumption is backed up by the existence of a patent regarding a possible implementation of ME [12]. It states that when a move elimination takes place, an entry is allocated in a structure, the *Multiple Instantiation Table* (MIT), which has only a few entries (e.g., 8). We will describe the MIT in more details in Section 4

[2]Moves from 32-bit registers can be eliminated since x86_64 states that the most significant 32 bits of the larger 64-bit register are set to zero on a 32-bit register-to-register move. This is not the case for 16- and 8-bit moves.

3. SPECULATIVE MEMORY BYPASSING

Background.

Another optimization leveraging register sharing can be applied to store-load pairs that fit within the instruction window. Instead of going through the Store Queue (SQ) and copying the data of a store to the destination register of a dependent load, it is much more efficient to rename the destination register of the load to the source register of the store, i.e., perform Speculative memory Bypassing (SMB) directly through the Physical Register File (PRF) [3]. An alternative is to use a dedicated buffer for SMB [5], which alleviates the need for register reference counting, but does not leverage the presence of the load data in the PRF. Bypassing can also be applied to load-load pairs [7, 13] by renaming the destination register of the second load to the destination register of the first one. This allows a register that was first bypassed through a store to continue feeding redundant loads even though the initial store has left the instruction window.

It has previously been argued that SMB brings marginal speedup if the memory dependency predictor is very accurate [4]. However, we point out that first, considering load-load pairs increases potential by allowing a single register to propagate for a longer time [7].

Second, [4] assumes the store-to-load forwarding (STLF) latency is a single cycle. In modern processors, the STLF latency is often not disclosed. But it is very unlikely to be a single cycle due to the necessity to 1) CAM the whole SQ 2) Handle specific cases such as operations with different sizes 3) Arbitrate potentially multiple hits using age-based encoding 4) Drive the data out of the SQ. For instance, Agner Fog's well-known optimization guide [14] reports at least 3 cycles for AMD Jaguar (8 for Bobcat). Additionally, in the case where the STLF latency is higher than the L1 hit latency (e.g., Bobcat [14]), a scheduling mispeculation[3] [15] will most likely take place every time store data gets forwarded to a load. SMB could prevent this from happening.

Finally, a larger instruction window increases potential by allowing to bypass from older instructions. [4] considered a much smaller window than windows of current microarchitectures (128-entry vs. 192-entry for Intel Haswell and 224-entry for Skylake). As a result, SMB may yet be beneficial to "big cores" that are aiming at sequential performance.

An Improved SMB Implementation.

Ultimately, we choose SMB as a way to evaluate our framework for physical register sharing, because is has been extensively studied in the literature [3, 5, 6, 4]. Nonetheless, we still consider several improvements over existing SMB mechanisms.

First, we introduce the Instruction Distance predictor, in-

[3]Load dependents were speculatively scheduled assuming that the load was going to hit in the L1, but they will not find their inputs on the bypass network because the load has been delayed by STLF.

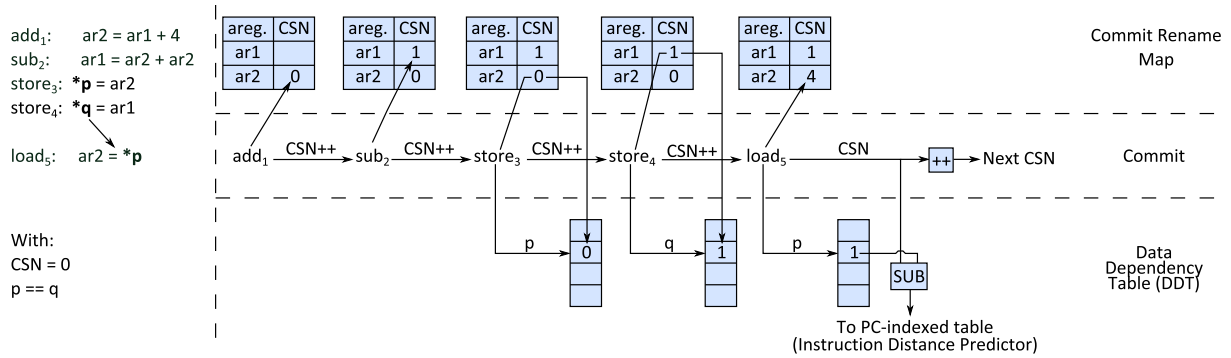


Figure 1: Computing the Instruction Distance of $load_5$ at Commit.

spired from the Store Distance predictor of Sha et al. [3]. This TAGE-like [9] predictor is accessed with the load PC and the global branch history, which are both available in the frontend. It predicts the distance between a load and the instruction producing the data that was read by the corresponding store. With this value, we can produce a ROB index and retrieve the bypass register identifier. We show that this predictor is more efficient at capturing SMB potential than the predictor used in NoSQ [3].

Second, we generalize SMB to load-load pairs, allowing a single physical register to be shared for a longer duration. In addition to uncovering more potential performance, this allows to stress the register sharing mechanism even more.

Third, indexing in the ROB allows bypassing from recently committed instructions in addition to inflight ones. As a result, we can study the impact of allowing SMB to span beyond the instruction window.

3.1 Instruction Distance Predictor

Computing the Distance.

Figure 1 depicts how the Instruction Distance predictor is able to capture the dependency between a load and an instruction producing the data sourced by the corresponding store. The predictor has two components: the Data Dependency Table (DDT) identifies pairs of instructions after retirement, while the actual predictor provides a distance to each load instruction in the frontend. Only the DDT is shown in the Figure. In the example, two stores to the same virtual address (but through different pointers) are followed by a load from that address. The load is dependent on $store_4$ and therefore uses the register produced by sub_2.

The dependency is identified in two steps. First, at Commit, the *Commit Sequence Number* (CSN) of register-defining instructions (e.g., add_1) is written in the Commit Rename Map entry corresponding to the destination architectural register. Second, when a store commits, it reads the Commit Rename Map for the CSN of the instruction that produced the data it stored. The DDT is then accessed using the virtual address the store wrote to and the CSN is written in the DDT entry.

As a result, the DDT will first record the CSN of add_1 (producer of $store_3$), whose value is 0, before recording the CSN of sub_2 (producer of $store_4$) since pointers p and q alias to the same location. At that time, the CSN in the DDT entry is 1. Then, when $load_5$ commits, it will read the entry corresponding to pointer p and find the CSN of sub_2, i.e., 1. Since the CSN is incremented every time an instruction commits, the current CSN for $load_5$ is 4, and the Instruction Distance can be computed by subtracting the current CSN with the one retrieved from the DDT. Finally, $load_5$ also puts its CSN in the Commit Rename Map as it produces $ar2$.

To generalize SMB to load-load pairs, the CSN of load instructions simply has to be put in the DDT after the old CSN has been read from the entry (not shown in Figure 1). Consequently, the Instruction Distance predictor can seamlessly handle load-load pairs.

Predicting the Distance.

The identified dependency is then transferred to the second component of the prediction infrastructure: the actual predictor that will be accessed by subsequent instances of the load in the frontend. We consider two predictors. First, one similar to the one used in NoSQ [3]. That is, we implement two tables, one is indexed via the load PC only, and the second via a hash of the load-PC, the global branch history and the path history[4]. If both hit, then the path-indexed table provides the prediction. On a misprediction, an entry is allocated in both tables. A confidence counter is associated to each entry, and SMB is only performed if the counter is saturated. The counter is incremented on a correct prediction, and reset to 0 if the distance does not match. Since mispredicting is costly (vs. not predicting at all), counters saturate at 15 (4-bit counters). Note that we only consider a predictor similar to NoSQ's, but not the whole NoSQ infrastructure.

Second, we consider a TAGE-like [9] predictor that features one direct mapped table (the base component) and five partially tagged components that are indexed using the PC, the global branch history, and the path history. We mix respectively 2, 5, 11, 27 and 64 bits of global branch history with 16 bits of path history and the PC to index the different components. The number of entries in each table are respectively 4096 (base table, 5b tag), 512 (10b tag), 512 (10b tag), 256 (11b tag), 128 (11b tag) and 128 (12b tag). Since the distance cannot be greater than the ROB size (plus instructions in flight from Rename to Dispatch, i.e., less than 256 instruc-

[4]Since we use large tables for this predictor, we XOR 8 bits of the global branch history with 8 bits of the path history, and XOR them with the load's address left-shifted by 4 bits.

tions total in our framework), 8-bit distance fields are sufficient, and 4-bit confidence counters are used, as previously. This predictor requires 12.2KB of storage, to be compared to the 17KB we allocate to the predictor of Sha et al. [3] (4K-entry for each table, 5b tag). We consider asymmetric sizes to illustrate the fact that more refined algorithms are required for SMB to be efficient, as we will show that our TAGE-like scheme outperforms the more conventional predictor in most cases. To that extent, we consider a very large 16K-entry DDT as a first design point.

Nonetheless, we found that using a 8.6KB DDT (1K entries containing the 64-bit virtual address[5] a 5b tag) yields an IPC that is within **2.2%** (0.7% gmean) of using an unlimited DDT, except in one benchmark out of 36 (*hmmer*, speedup is 1.03 vs. 1.17). As a result, the overall cost of this embodiment of SMB would be around 21KB.

Since this paper's first goal is to introduce a new register sharing scheme, we did not tune our Instruction Distance Predictor for memory dependency prediction, hence we still use a discrete memory dependency predictor (Store Sets [16]). As a result, SMB can improve performance because 1) The STLF latency may be hidden 2) False dependencies generated by the Store Sets predictor can be removed by bypassing the register 3) Dependencies missed by the Store Sets predictor can be identified by SMB, reducing the number of memory traps (RAW memory violations).

Nonetheless, it is our intuition that the Instruction Distance Predictor could also act as the memory dependency predictor, although tuning may be required. This would help amortize the cost of the SMB infrastructure and this approach is in fact used in NoSQ [3], although only store-load SMB is considered in that case. We leave the design of a TAGE-like memory dependency predictor for future work.

3.2 Breaking Memory Dependencies Through the Renamer

The distance predictor identifies the instruction producing the result of a load. The Reorder Buffer (ROB) can be used to retrieve the identifier of the physical register produced by this instruction. Once a pair is identified using the Instruction Distance, we simply rename the destination of the load instruction to the destination of the matching instruction.

Figure 2 depicts how this flavour of SMB processes a simple snippet of code. First, sub_1 is allocated an entry in the ROB at Dispatch (1). Then, when $load_3$ is fetched or decoded, it accesses the Instruction Distance predictor to retrieve how far the instruction producing the data it is expected to load is, if there is one (2). If the Instruction Distance falls within the bounds of the ROB, the physical register identifier of the producing instruction is retrieved (3). In particular, the instruction of interest is either:

i In flight between Rename and Dispatch. In that case, we consider that the physical register identifier is reachable from a queue of instructions pending allocation in the ROB.

ii In the ROB. The ROB entry is accessed and the register identifier retrieved.

iii Already out of the ROB as it has committed. No speculation takes place in that case.

Finally, the identifier is used by the Renamer as the new destination register for the load's architectural destination register. This replaces the chain of three dependencies between sub_1 and add_4 by a single register dependency (4).

Validation.

Validation is done by reading the bypassed register when the load issues, and comparing it against the data coming from the memory hierarchy at Writeback. Consequently, it is required that the bypassed register be marked as a source of the load instruction. This may be problematic since the "free" dependency of load instructions is generally used to implement memory dependency predictions. That is, in the scheduler, load instructions may have an extra source operand that corresponds to the store they are predicted to depend on. This is not problematic since loads have one less source operand than stores by construction.

However, since we require loads to depend on yet another instruction, we must make sure that this additional dependency can co-exist with the memory dependency prediction in the scheduler. The most straightforward approach would be to use the Distance predictor as the memory dependency predictor, as in NoSQ [3]. Yet, because we do not implement the whole NoSQ infrastructure and since we did not tune the Distance predictor to handle memory dependency predictions well, we assume the presence of enough sources in the scheduler, which is optimistic but not unrealistic. For instance, AMD Bulldozer's scheduler supports 4 sources per instruction [17], which would be enough for both additional dependencies.

Regardless, this also implies one additional read port on the register file per issued load per cycle. Fortunately, since not all loads are bypassed, and many modern microarchitectures can only issue two loads per cycle at most (e.g., Intel Haswell), the overhead remains limited.

3.3 Bypassing from Committed Instructions

In the previous paragraphs, we stated that once the Instruction Distance of a load is predicted, the instruction of interest can be in three distinct states: pending allocation in the ROB, in the ROB, and committed. In the first two cases, SMB can take place, while in the third case, nothing can happen since the instruction has left the ROB.

However, bypassing from recently committed instructions is possible provided that we guarantee that the physical register is still valid. For this purpose, we propose a simple modification of the commit phase in the pipeline. Indeed, a possible implementation of the ROB is to use a circular buffer, with older instructions at the *head* and younger ones at the *tail*. Two pointers are responsible for delimiting occupied entries from free entries, that is, when an instruction is added, the tail pointer is incremented, while the head pointer is incremented when an instruction commits. At the same time as

[5]Caching the – usually smaller – physical address is possible and preferable if SMT is implemented, but this requires one or more TLB ports to update the DDT, or caching translations in the LQ/SQ until instructions update the DDT.

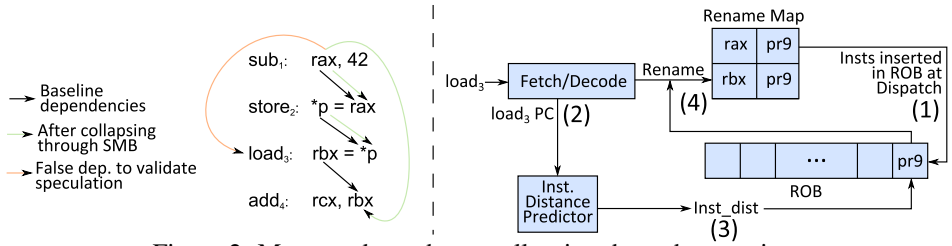


Figure 2: Memory dependency collapsing through renaming.

an instruction commits, it also releases the previous physical register mapped to its architectural register target. However, data (register numbers, etc.) remain in the ROB entry till the entry is overwritten, i.e., until a new instruction is added to the ROB and the tail pointer has reached this entry.

Therefore, we simply propose to delay the reclaiming of registers after commit as late as possible. A third pointer – *release_head* – that points to the oldest retired entry containing valid state is used for the register reclaiming process. In this context, the destination register identifier of the instruction contained in the ROB entry is guaranteed **not** to have been overwritten and can therefore be bypassed.

We reclaim registers **lazily**, e.g., when the Free List signals that only a few free registers are available or when the ROB is almost full i.e., the *tail* pointer approaches the *release_head* pointer. In that event, post-commit hardware scans the ROB starting from the *release_head* pointer and frees physical registers corresponding to old versions of architectural registers redefined by the committed ROB entries.

Using this mechanism, there is more potential for SMB since registers belonging to committed instructions are still reachable. As the size of the ROB grows, potential should increase even more since more committed state can be kept.

Our proposal should be contrasted with NoSQ [3]. In that proposal, the *Store Register Queue* (SRQ) has to remove store source registers as stores are committed, by construction. Indeed, once a store commits, its source register may be reallocated to another instruction in the near future, and it cannot be bypassed safely anymore. Therefore, using the ROB for SMB can uncover more potential.

4. REGISTER REFERENCE COUNTING

Both ME and SMB through the register file require register reference counting to avoid reclaiming a live register. In general, per-register counters are considered, yet we argued in Section 1 that they might not be practical enough, in particular during pipeline flushes. In this Section, we first describe how the Renamer state can be rolled back on a pipeline flush. Then, we review three previously proposed register sharing schemes and illustrate their shortcomings. Finally, we propose a new scheme that can track enough registers to not hold back ME and SMB while requiring limited storage and being checkpointable.

4.1 Recovery in a Regular Register Renamer

A commonly depicted renamer consists of three structures: 1) The Rename Map (RM), that maps architectural registers to physical registers 2) The Free List (FL), a circular buffer that contains free physical registers and 3) The *Commit* Rename Map (CRM), that contains non-speculative mappings. When an instruction is renamed, it first looks for the physical register corresponding to its source architectural registers in the RM. Then, it takes a free register at the head of the FL and places it in the RM entry corresponding to its architectural destination register.

Assuming a checkpointing architecture where the RM and FL pointer are checkpointed (e.g., at every low confidence branches [18]), recovering from a pipeline flush implies copying the checkpointed RM to the processor's RM and restoring the FL pointer. For pipeline squashes taking place at Commit (e.g., value mispredictions [19]), copying the CRM in the RM and restoring the *Committed* FL pointer are sufficient (i.e., no checkpointing is necessary).

As a result, recovering the renamer state on a branch misprediction is fairly simple and can be done in a single cycle if a checkpointing architecture is implemented [20, 21]. Unfortunately, allowing registers to be shared complicates this process. For instance, per-register counters cannot be checkpointed, by construction. This is problematic since the branch misprediction recovery latency has been shown to strongly contribute to deeply pipelined processors performance [8]. As a result, there is a call for a sharing scheme that would allow such a simple recovery process for the renamer state.

4.2 Previous Proposals for Reference Counting

2D Matrices.

Roth [22] describes two ways to count register references. First, through a 2D matrix whose columns correspond with physical registers and rows correspond with ROB entries. A set bit means that the ROB entry corresponding to the row references the physical register corresponding to the column. As a result, a register is free if the OR of all rows returns 0. However, this design appears impractical for a modern processor such as Intel Haswell (192-entry ROB, 168 INT regs, 168 FP/SIMD regs). The matrix would require a $2 \times 192 \times 168$ bits, amounting to around 7.8KB of storage (not even counting the columns dedicated to the Commit Rename Map). As a comparison, the matrix of a baseline matrix scheduler the size of Haswell's would require 0.44KB only. One could argue that 7.8KB is not so high a cost to pay for reference counting. This may be true now, but due to its matrix nature, we would argue that it is intrinsically not scalable anyway.

Battle et al. improve on this scheme by diminishing the matrix requirements [23].The matrix only requires *#preg* $\times$ *max_sharers_per_register* bits, but is entirely checkpointed in a processor using checkpoints, i.e., checkpoint storage is

still significant.

Reference Counters.

Second, references can be counted through implementing one counter per physical register and incrementing it on allocation or re-reference[6] while decrementing it on regular deallocation and de-reference. The value of the counter gives the state of the register: if the counter is 0, the register is free, otherwise, it is not. Counters appear as the most intuitive way to count references. However, a problem arises for squashes.

In a checkpointing architecture that allows register sharing, reference counters need to be rolled back in addition to the RM and the FL pointer. This is problematic because their value cannot simply be checkpointed and restored. The reason is that a counter can have been decremented when an instruction older than the checkpoint committed. Restoring the checkpointed value would consider that the older instruction has not committed, and the pointed physical register would therefore never be freed.

As a result, squashing implies walking the squashed instructions and decrementing reference counters. That is, the pipeline cannot restart immediately because the ROB has to be walked sequentially (potentially by chunks). While this might have little to moderate impact on performance [18], it complicates the recovery process and should be avoided for deeply pipelined processors [8].

Enabling fast recovery for reference counters would imply 1) Updating the most recent checkpoint each time an instruction references a register and all checkpoints each time an older instruction commits 2) Having enough storage in each checkpoint for every register of the processor (i.e., at least a few bits for each of the 336 registers of Haswell). While updating checkpoints on the fly may be doable, increasing the size of each checkpoint by 600+ bits may not be desirable.

Multiple Instance Table (MIT).

The MIT, introduced in an Intel patent of Raikin et al. [12], allows potentially limited register sharing capacity. It appears as small fully-associative structure of which an entry is allocated when a move elimination candidate is encountered. Each entry contains a bit-vector corresponding to architectural registers, e.g., if bits 0 and 1 are set, then both *rax* and *rbx* map to the physical register. A bit is reset when the corresponding architectural register is redefined, and if the whole vector is zero, then the physical register is freed.

However, the MIT leverages the property of move elimination that both architectural registers pointing to the physical register are known since they are visible in the move instruction. This is not the case for SMB as the source architectural register of the store has potentially already been re-renamed when the matching load is renamed (store-load case). In other words, the MIT cannot track sharing in SMB, because the algorithm is based on architectural names. Moreover, it requires more checkpoint storage per entry than the scheme we propose (#arch_reg bits per entry).

[6]For instance, a load that uses its producer store's source register as its destination register re-references the register.

Register Duplicate Array (RDA).

Lastly, the RDA is described in an Apple patent of Sundar et al. [24]. Similarly to the MIT, it is a small fully associative structures where entries are allocated and de-allocated on-demand. Each entry contains a reference counter which tracks the number of sharers. As a result, contrary to the MIT, it is not limited to move elimination. However, to make the structure checkpointable, all counters (in the RDA and checkpointed) relating to the physical register of interest must be updated when an instruction retires. In other words, committing a mapping relating to a tracked physical register requires decrementing up to n counters, with n the number of checkpoints.

4.3 Limited Reference Counting

4.3.1 A Single Counter Isn't Enough

We previously stated that the value of a reference counter cannot be checkpointed since the number of sharers is modified by the Commit stage. However, we observe that by using two counters per register instead of one, checkpoints can be used. A first counter, *referenced* is incremented when a register is bypassed (i.e., not allocated from the Free List). For instance, in the case of SMB, an instruction feeding a store will be allocated a register, but the *bypassed* counter will not be touched. It is only when the bypassing load enters Rename that it will be incremented, since there is one more bypassing instruction referencing this register.

A second counter, *committed*, is increased when an instruction that overwrites one of the architectural register referenced by the physical register commits. Given the previous example, the *committed* field will be incremented when an instruction that writes the same architectural register as either the bypassing load or the instruction feeding the associated store is committed.

It follows that if *referenced* is equal to *committed*, then the register can be freed by the next committing instruction that overwrites the mapping containing the physical register. At that time, both counters will be reset. In the remainder of the paper, we refer to *committed* and *referenced* as counters, but it should be noted that those structures are really up-counters that can be reset, i.e., they are never decremented.

Checkpointing.

This dual-counter scheme lends itself to checkpointing because it differentiates between the number of committed references and the total number of references (which includes committed references). Indeed, the number of committed references is architectural, therefore, it is always correct. Only the total number of references may be stale after a pipeline flush, if squashed instructions did reference the register on the wrong path.

As a result, checkpointing the *referenced* field only allows to restore the reference counting mechanism to a correct state straightforwardly on a pipeline flush: Checkpointed values simply have to be restored. There is however a specific case where additional steps must be taken to ensure proper recovery, but this does not increase recovery latency, as we illustrate in the following example.

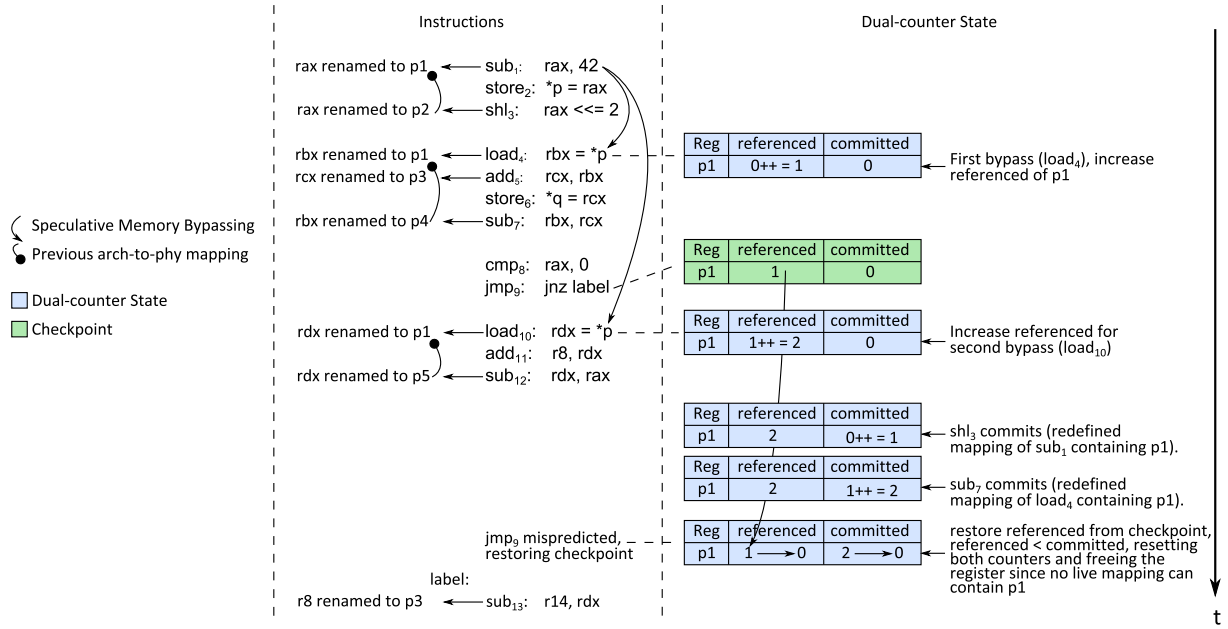


Figure 3: Register Reference Counting using the Inflight Shared Registers Buffer.

Working Example.

Figure 3 provides an example of how this dual-counter scheme is able to track how many times a register is bypassed in the presence of a branch misprediction. While it can handle both SMB and ME, we focus on SMB to give a non-trivial example. In the Figure, the *referenced* counter of register $p1$ is incremented to 1 when load_4 hits $p1$ in the ROB. Subsequently, a branch, jmp_8, is encountered and *referenced* is checkpointed. Past the predicted non-taken branch, load_{10} also hits $p1$ in the ROB and *referenced* is incremented to 2.

At some point, shl_3 and sub_7 are committed. Since both overwrite mappings containing p1 (respectively $rax => p1$ of sub_1 and $rbx => p1$ of load_4), the *committed* counter of $p1$ is incremented two times and reaches 2, the same value as *referenced*. This means that when the instruction that overwrites the mapping of load_{10} ($rdx => p1$) commits, it will free $p1$ and reset both counters.

Unfortunately, jmp_8 was mispredicted. To recover, the state contained in the checkpoint for $p1$ is copied to the *referenced* field, correctly reflecting the fact that there is only one instruction using $p1$ through SMB (load_4). However, since *committed* is 2 (i.e., greater than *referenced*), this means that the last instruction that overwrote a mapping containing $p1$ (i.e., sub_7) should have freed $p1$ and reset both fields.

Consequently, an additional step is required to compare the current *committed* value to the *referenced* value of the checkpoint. If *committed* is strictly greater, the register is freed and both counters are reset. Otherwise, one or more subsequent instructions have a reference to an old mapping containing the register, which will therefore be freed later.

In any case, these operations are quite simple and process narrow values. Therefore, we expect recovery to take a few cycles at most and to be doable in a single cycle. In other words, restoring a correct state can be done in a single cycle. Pushing the registers freed during recovery to the Free List may be done in subsequent cycles, but the processor need not wait for this before restarting Fetch.

4.3.2 Not All Registers Are Shared

We base our final reference counting scheme on the intuition that if a snapshot of the register file had to be taken, most would either be referenced one time only or be free, and a few would be referenced more than one time. Ultimately, the additional state that needs to be checkpointed corresponds to the latter ones, since restoring the Free List pointer covers the ones that are referenced only once.

As a result, we introduce the *Inflight Shared Register Buffer* (ISRB). Much like the MIT [12] and RDA [24], the ISRB is a small fully-associative buffer that tracks registers that have currently more than one sharer. Each entry of the ISRB comprises the physical register identifier, that acts as the tag, and the two counters, *referenced* and *committed*.

Bypass Potential.

Being able to track a small number of registers has implications on how bypassed registers are processed. First, when a bypassing event is detected (e.g., move elimination or SMB), an ISRB entry is allocated for the physical register. If the ISRB is full, bypassing does not take place. Fortunately, we will show that only 16 to 32 entries are sufficient to capture most of the potential of both ME and SMB.

Register Reclaiming.

The register reclaiming hardware has to CAM the ISRB when it attempts to free every register. On a CAM match, if both counters are equal[7], then the register and the ISRB entry are freed. In that event, all checkpointed *referenced* counters

[7]This can be checked easily by computing $\text{NOR}_{bitwise}(referenced\ \text{XOR}_{bitwise}\ committed)$. Since the fields are small (we found 3 bits to be sufficient), this should be fast and quite cheap.

Front End	L1I 8-way 32KB, 1 cycle, Perfect TLB; 32B fetch buffer (two 16-byte blocks each cycle, potentially over one taken branch) w/ 8-wide fetch, 8-wide decode, 8-wide rename TAGE 1+12 components [9] 15K entry total, 20 cycles min. mis. penalty; 2-way 4K-entry BTB, 32-entry RAS;
Execution	192-entry ROB, 60-entry IQ unified, 72/48-entry LQ/SQ (STLF lat. 4 cycles), 256/256 INT/FP; 4K-SSID/LFST Store Sets, not rolled-back on squash[16]; 6-issue, 4ALU(1c), 1Mul-Div(3c/25c*), 2FP(3c), 2FPMulDiv(5c/10c*), 2Ld/Str, 1Str; Full bypass; 8-wide retire
Caches	L1D 8-way 32KB, 4 cycles, 64 MSHRs, 2 load ports; Unified L2 16-way 1MB, 12 cycles, 64 MSHRs, no port constraints, Stride prefetcher, degree 8, distance 1; All caches have 64B lines and LRU replacement;
Memory	Single channel DDR3-1600 (11-11-11), 2 ranks, 8 banks/rank, 8K row-buffer, tREFI 7.8us; Across a 64B bus; Min. Read Lat.: 75 cycles, Max. 185 cycles.
Instruction Distance Predictor	**NoSQ:** 4K(d. map) + 4K(hash) [3], 5b tag, 4b conf., 17KB **TAGE-like:** 1 + 5-comp. [9] (5.25K entry total), 12.2KB **DDT Base:** 16K-entry DDT (14b tag),156KB; **DDT Opt.** 1K-entry DDT (5b tag), 8.6KB

Table 1: Simulator configuration overview. *not pipelined.

corresponding to this entry are also gang-reset. Otherwise, the *committed* counter is incremented. On a mismatch, the register is freed.

Checkpoint Recovery.

Checkpoint recovery simply requires the *referenced* field of the checkpoint to be copied to the corresponding ISRB entry. In parallel, the *committed* field is pulled from the ISRB and checked against the checkpointed *referenced* field. If *committed* is greater than *referenced*, the physical register and the ISRB entry are freed, as previously. In addition, if both counters are zero, the ISRB entry is freed, as the physical register has either been freed by restoring the Free List pointer (re-allocation and reuse are younger than the checkpoint) or will be freed when committing an instruction older than the checkpoint (re-allocation is older than the checkpoint, but reuse is younger). If the ISRB entry is already free, nothing happens.

The only requirement for this scheme to ensure correctness is that when an ISRB entry is freed, all corresponding entries in the checkpoints be marked as invalid (i.e., have their *referenced* counter reset). This takes care of the particular case where an ISRB entry that was freed on the correct path is still tracked in a younger checkpoint. If said checkpoint is not invalidated when the ISRB entry is freed, a wrong *referenced* value may be restored if the entry has been re-allocated on the wrong path, allowing physical registers to "leak". By resetting checkpoint entries, a value of 0 will be restored for *referenced*.

4.3.3 Checkpointing Storage Overhead

We previously mentioned that each checkpoint should contain a very limited subset of the ISRB state: the *referenced* field. This amounts to *n*-bit counters times the number of ISRB entries. For an 8-entry ISRB and 3-bit counters, this amounts to 24 bits of storage, while a 16-entry ISRB requires 48 bits and a 32-entry ISRB requires 96 bits only. In comparison, saving the x86_64 Rename Map requires at least 256 bits ((16 GPRs + 16 SIMD registers) $\times$ 8-bit identifiers).

4.3.4 Complexity of the ISRB

Although it features a small number of entries, the ISRB is a heavily contended structure. Each cycle, it must be associatively searched by the renamer for each register that is a candidate for bypassing. In the meantime, the register reclaiming mechanism must also associatively search the ISRB for every register it attempts to free. This amounts to *rename_width* + *reclaim_width* CAM ports in the worst case. Fortunately, the number of ports needed by the renamer can be reduced since only bypassing candidates require a port at rename time.

One could also reduce the number of ports dedicated to register reclaiming through leveraging the fact that not every register is a likely candidate for sharing. That is, only specific instructions need to check the ISRB when they are processed by the register reclaiming mechanism.

By writing a flag in the Rename Map at Rename, and propagating this flag to the instruction that redefines the register, we can avoid many ISRB accesses at reclaiming time. For instance, for ME, the bit is set for both the architectural source and destination registers when the move is eliminated. For SMB, all loads set the bit for their architectural destination register, while all stores set the bit for the architectural register that contains store data. Any other instruction resets the flag of their destination register at Rename.

Then, if an instruction redefines a register whose flag is set, it will check the ISRB for the physical register previously mapped to the architectural register at reclaiming time. Otherwise, the old physical register can be freed without accessing the ISRB.

5. EVALUATION METHODOLOGY

5.1 Simulation Infrastructure

We use the *gem5* cycle-level simulator [25] implementing the x86_64 ISA. We consider a relatively aggressive 4GHz, 6-wide issue pipeline. The fetch-to-commit latency is 19 cycles. The in-order front-end and in-order back-end are overdimensioned to treat up to 8 μ-ops per cycle. We model a deep front-end (15 cycles) coupled to a shallow back-end (4 cycles) to obtain a realistic branch misprediction penalty (20 cycles minimum). We allow two 16-byte blocks of instructions to be fetched each cycle, potentially over a single taken branch. Table 1 describes the characteristics of the baseline pipeline we use in more details.

In our experiments, the store-to-load forwarding latency is equal to the L1 latency (4 cycles), and only loads that are fully or partially contained in inflight stores (e.g., load 32 bits from a 64-bit store) can read data from the Store Queue. Overlapping loads wait for the stores they overlap with to write back.

Lastly, in the version of gem5-x86 we use, all register-to-register moves are *merge* μ-ops, meaning that a 64-bit move depends on both its source operand and its destination operand [26]. The latter dependency is not required because the whole destination is overwritten (also the case for 32-bit move). However, if it is not removed, the gains of ME will be overestimated. Consequently, we reimplemented 64 and 32-bit register-to-register moves as true moves while 8- and 16-bit moves remain *merge* μ-ops.

5.2 Benchmarks

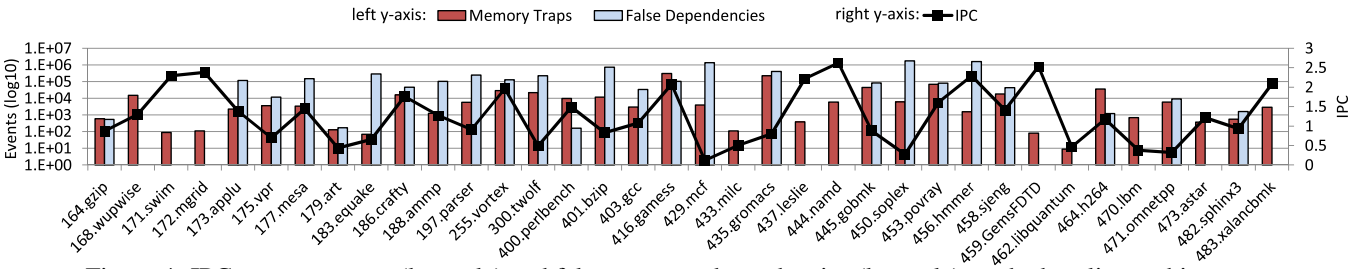


Figure 4: IPC, memory traps (logscale) and false memory dependencies (logscale) on the baseline architecture.

We use a subset of the SPECCPU 2000 [27] and SPECCPU 2006 [28] suites to evaluate our contributions. Specifically, we use 18 integer benchmarks and 18 floating-point programs[8]. We use the *reference* inputs provided in the SPEC software packages. To get relevant numbers, we identify a region of interest in the benchmark using Simpoint 3.2 [29]. We simulate the resulting slice of 150M instructions by warming up the caches and branch predictor for 50M instructions then collecting statistics for 100M instructions.

Figure 4 reports the IPC of each benchmark on the baseline pipeline configuration, as well as the number of memory order violations (traps that result in a pipeline flush in *gem5*) and the number of false dependencies introduced by the Store Sets predictor. The two latter are reported on a log scale.

6. EXPERIMENTAL RESULTS

6.1 Impact of ISRB Parameters on ME

Impact of the Number of Entries.

Figure 5 (a) illustrates the fact that only a few entries are sufficient to benefit from the speedup brought by move elimination. In this case, we only implement ME for 64- and 32-bit integer register to integer register moves.[9] Nonetheless, even limited to integer registers, ME is able to increase performance significantly, up to 5.2% in *crafty*. As expected, the gain is generally lower for FP benchmarks.

Moreover, we observe that a if 32-entry ISRB is enough to benefit from almost the whole potential of ME, 16 entries are generally sufficient, and 8 entries perform reasonably well. In any case, speedups are generally limited (1% geometric mean), which is in accordance with what Jourdan et al. observed [1].

Figure 5 (b) illustrates the percentage of renamed instructions that were eliminated thanks to move elimination (assuming an unlimited ISRB). These instructions are put in the ROB and renamed, but they do not occupy an entry in the out-of-order scheduler or an ALU since they are never issued. We note that this proportion does not correlate strongly with the performance gain. For instance, in *vortex*, almost 10% of the renamed instructions are eliminated, but the gain is less than 3%. Conversely, in *namd*, only 5% of the renamed instructions are eliminated, but this allows to increase performance by around 4.5%.

[8]We do not use the whole suites due to some currently missing system calls/x87 instructions in our version of *gem5-x86*.

[9]Recent Intel microarchitectures also implement it for floating point moves and moves with zero extend [2].

6.2 Impact of ISRB Parameters on SMB

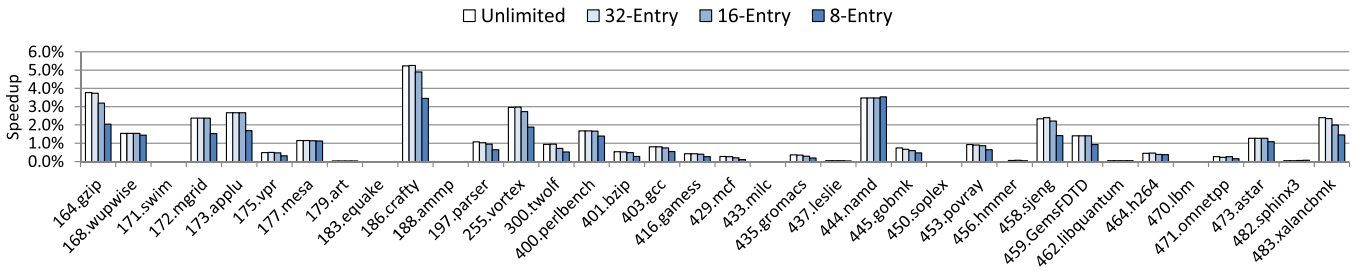
Number of Entries.

Figure 6 (a) considers the performance gain brought by SMB for both store-load and load-load pairs (bypassing from non-committed instructions only) depending on the number of ISRB entries. Contrary to move elimination, SMB puts more pressure on the ISRB, hence 24 entries are required to achieve almost the full potential of SMB. By lack of space, we do not report numbers for a limited-size DDT, but we found IPC to be within 3% of using an unlimited DDT, except in *hammer* where speedup would be 3% instead of 17%. However, the use of an associative/skewed table is possible to address this drop. We also note that using a 2-table predictor like NoSQ's does not improve performance much, contrarily to our TAGE-like predictor.

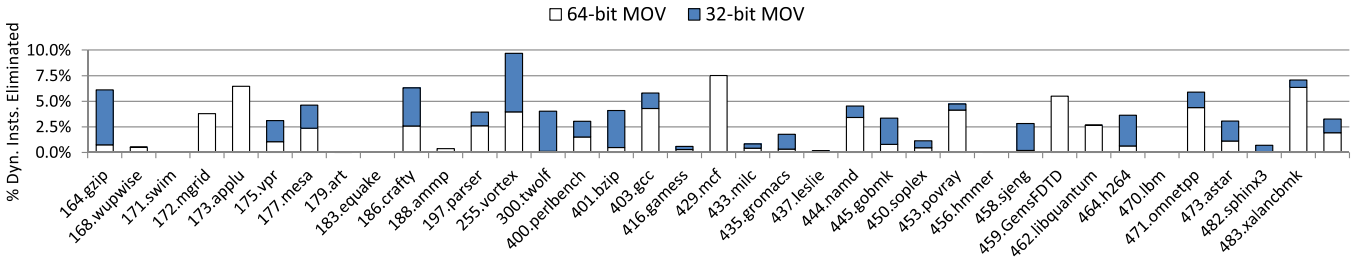
In addition, Figure 6 (b) reports the reduction in number of memory order violations and false memory dependencies when using an unlimited ISRB and our TAGE-like predictor. Those are reported only for benchmarks where those events occur reasonably often in the baseline depicted in Figure 4 (i.e., at least 1K memory traps, and at least 10K false dependencies). We can observe a correlation between the speedup and the reduction of at least one type of event (e.g., *wupwise, applu, bzip, gamess, gromacs* and *hmmer*), hinting that SMB itself may not improve performance, but rather, the TAGE-like Distance Predictor is able to capture dependencies that Store Sets cannot. Yet, in *astar*, performance increases but few memory traps and false dependencies take place. In this case, performance increases because the STLF latency is hidden by SMB (i.e., bypassed loads have a latency of 0 instead of 4 cycles).

This is not incompatible with [4], as if we had a perfect memory dependency predictor, then the only source of speedup would be hiding the STLF latency, and speedups would be lower. Yet, since Store Sets is not a perfect predictor, we obtain good speedups in several benchmarks. As a result, those numbers can be seen as a hint that TAGE-like memory dependency prediction may be worthwhile rather than a case for SMB itself.

Lastly, we also considered bypassing only from stores, and found that in a few cases, this was particularly detrimental. By lack of space, we do not report all bars, but specifically, notable decreases include *astar* (2.8% vs. 16.6%), *wupwise* (18.7% vs. 21.9%), *applu* (29.6% vs. 35.3%), *bzip* (2.0%

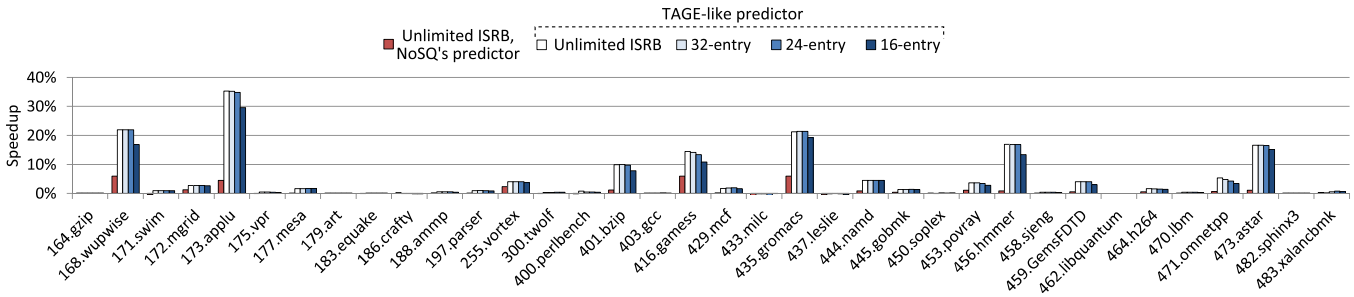


(a) Varying the number of entries.

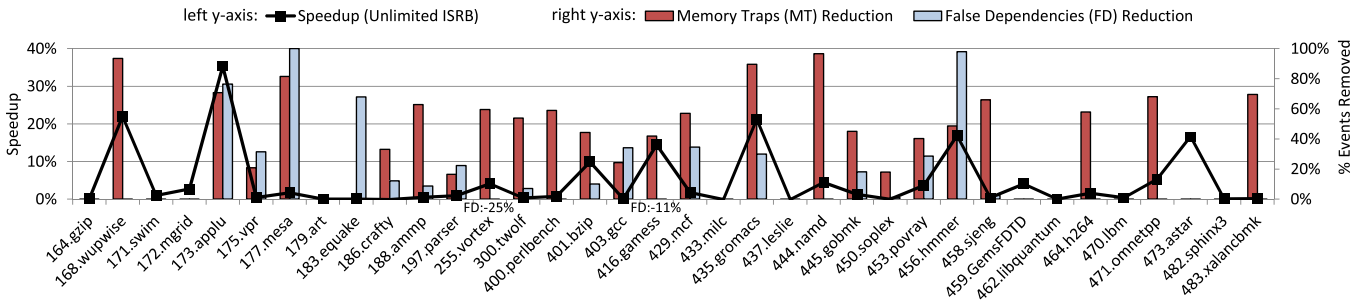


(b) Percentage of renamed instructions that were eliminated (Unlimited ISRB).

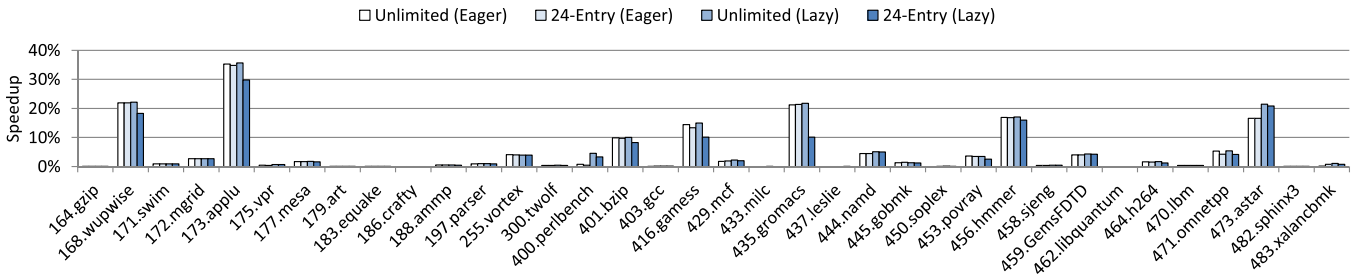
Figure 5: Impact of ISRB parameters on the performance gain (speedup over baseline) brought by move elimination.



(a) Varying the number of entries (Baseline SMB)



(b) Correlating the performance gain of SMB with the reduction in memory traps/false memory dependencies.



(c) Varying the number of entries (eager reclaim vs. lazy reclaim).

Figure 6: Impact of ISRB parameters on the performance gain (speedup over baseline) brought by SMB.

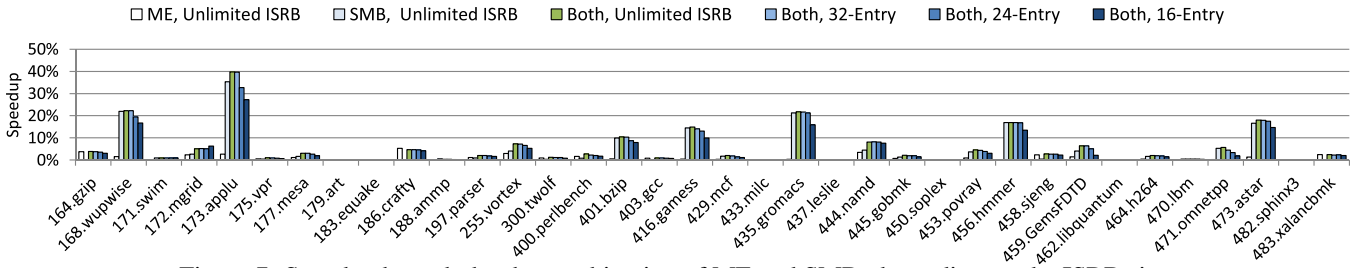


Figure 7: Speedup brought by the combination of ME and SMB, depending on the ISRB size.

vs. 9.8%) and *hmmer* (7.5% vs. 16.9%). The distance predictor accuracy is similar in both cases. This suggests that in those benchmarks, a fair amount of the speedup actually comes from hiding the latency of loads, which can be done more often if load-load bypassing is allowed.

Bypassing from Committed Instructions.

Figure 6 (c) extends SMB to committed instructions that still occupy an entry in the ROB. Register reclaiming is triggered when the Free List signals that only $rename\_width \times 2$ free registers are left or when *head* reaches *release_head*. We consider an unlimited ISRB and a 24-entry ISRB. The Figure shows that allowing to bypass from recently committed instructions generally has a marginal impact, even though the average percentage of retired loads that were bypassed increases from 32.3% to 35.7% (Unlimited ISRB).

This is not unexpected. Indeed, when bypassing from committed instructions, only the L1 latency can be hidden because the Store Sets predictor cannot make a load falsely depend (or miss a dependency) on a committed store (strictly speaking, it can, but this has no performance impact). As a result, improvements are modest. We found this to be true even when the size of the ROB is increased to 512 and the ISRB size is kept unlimited.

Performance decreases significantly in some cases (e.g., *wupwise, applu, gamess* and *gromacs*) when the ISRB is of limited size. In that case, less bypassing takes place within the window because some ISRB entries are taken up for bypassing from committed instructions, reducing the amount of ISRB entries used to remove memory traps/false dependencies.

The only exception to this trend is *astar*. In this benchmark, Store Sets perform quite well (366 memory traps and no false dependencies reported for 100M instructions), yet, regular SMB provided 16.5% speedup, simply by reducing the latency of many loads (53.9% of the committed loads). As a result, it is not surprising that when the STLF latency can be hidden for more loads (59.5% of the committed loads), speedup increases to 20.8%. However, this behavior appears to be the exception, which is in accordance with [4].

Consequently, if this optimization allows to seamlessly bypass from committed instructions, it does not appear useful in the context of SMB, and we therefore focus on instructions that fit in the window in the remainder of this paper. Lazy reclaim might be considered worthwhile in the context of *Register Integration* [30], however.

6.3 Combined

Impact of the ISRB Size.

Figure 7 shows speedup over baseline when both SMB and ME are used. If we consider a 32-entry ISRB, performance is often higher than ME or SMB alone (using an unlimited ISRB). 24 entries only also appears as a good tradeoff overall since performance is close to the 32-entry version.

However, using only 16 entries often results in performance being worse than the best of either ME or SMB using an unlimited ISRB. The reason is that both ME and SMB share the ISRB, hence performance decreases in benchmarks that benefit from a single mechanism. An exception is *mgrid*, where less bypass mispredictions take place because of the limited ISRB, yet enough bypassing still takes place to improve performance slightly.

As a result, for smaller ISRB sizes, it might prove efficient to forbid a single mechanism from using the whole ISRB, e.g., by forbidding ME to be performed if less than n entries are available in the ISRB. We leave the exploration of such mechanism for future work as it depends on which particular combinations of optimizations are implemented.

Counter Width.

We found that using 3-bit counters is sufficient to get within 1.3% (worst case, gmean is 0.1%) of the ideal IPC using both SMB and ME.[10] Using a 32-entry ISRB with 3-bit counters yields an average speedup of 5.5% (geomean), while requiring 480 bits of CPU storage, and 96 bits per additional checkpoint. An unlimited ISRB with 32-bit fields yields an average speedup of 5.6%.

ISRB Accesses.

On our benchmark suite, considering both SMB and ME and a 32-entry ISRB, the average distance (in instructions) between two instructions that attempt to allocate an entry in the ISRB is 19.7 (min. is 3.8). As a result, provisioning less ports than the rename width should not be problematic, especially since SMB or ME can be aborted if not enough ports are available (stalling is not necessary).

Similarly, on our benchmark suite, the average distance between committing instructions that must check the ISRB is 3.4 (min. is 2.3). As a result, more ports are required for reclaiming, especially since those accesses *must* be made to enforce correctness. Since up to 53% (32% average) of the instructions needing access to the ISRB at reclaiming are im-

[10] ME actually performs well on all benchmarks but one when single-bit counters are used. This would facilitate recovery by removing the need to compare fields, and is therefore a valid design point if only ME is considered.

mediately followed by an instruction that also needs access to the ISRB, provisioning 6 ports for a commit-width of 8 should prevent Commit from stalling in the general case.

7. OTHER RELATED WORKS

7.1 Non Sharing-based Schemes

Tyson et al. [5] implement speculative memory bypassing for store-load pairs through a dedicated buffer, alleviating the need for register sharing, but not taking advantage of the fact that values are already in the PRF.

Önder and Gupta propose to associate a memory address to each physical register [13]. An issuing load can then check if a register already contains the value located at the address it is accessing before accessing the L1 Cache. *Silent* stores can also be eliminated through this technique. Unfortunately, this technique would be very expensive to implement in a modern superscalar processor, as each issued memory instruction must associatively access #*phys_reg* virtual addresses. For instance, on Haswell, the PRF has 168 INT and 168 FP/SIMD registers and three addresses can be generated each cycle.

Similarly, Yang and Gupta [31] describe a mechanism where load and store reuse is attempted before sending memory instructions to the cache. The main difference with Onder and Gupta [13] is that bypassing is done through a PC-based direct-mapped table rather than by associatively searching the whole register file on every access. To predict, the load PC is used to index the *Load Table* (LT) before the cache access. The LT entry contains an address, and on a match, the value in the entry can be reused (same-load reuse). Otherwise, another table is accessed to try *different-load reuse*. This table contains a pointer to another entry of the LT. If this fails, the same process is attempted by accessing another table containing pointers to the *Store Table* (*store-load reuse*). The cache is accessed if none of the three reuse types can take place. Note that to identify *different load reuse* and *store-to-load reuse* (i.e., train the tables containing the pointers to the LT and ST), associatively searching the LT and ST on the address field is still required.

Sodani and Sohi propose *instruction reuse* in which instructions (including loads) can produce their result faster by checking in the *Reuse Buffer* (RB) if its operands correspond to past operands [32]. If it is the case, the result is directly available from the RB. As in the two previous schemes [13, 31], the RB must be kept coherent with memory.

7.2 Sharing-based Schemes

Jourdan et al. [1] generalize register sharing by comparing the result of an instruction to recently produced results contained in a buffer. On a match, the new instruction can be renamed to the register containing the recently produced value. However, since this comparison must be done before renaming takes place, it requires that the younger instruction be value predicted, a load immediate, or that sharing be added on top of instruction reuse [32] or Virtual Registers [33]. Register sharing is achieved through counters.

Petric et al. propose RENO, a rename-time hardware optimizer [34]. RENO is able to perform move elimination and *non-speculative* memory bypassing, amongst others. Yet, it relies on per-register counter for register sharing. Fortunately, since RENO is able to dynamically eliminate up to 22% of the dynamic instructions of SPECint2000 (i.e., not all registers are shared), we believe that using the ISRB would not hinder its performance.

Fahs et al. propose Continuous Optimization (CO) [35]. CO is similar to RENO in the optimizations that it can perform dynamically, and also uses per-register counters to implement register sharing. As for RENO, using a limited counting scheme such as the ISRB should be straightforward.

Finally, Petric et al. [30] propose an optimized implementation of *register integration* that addresses the same inefficiencies as instruction reuse [32] and the generalized register sharing scheme of Jourdan et al. [1]. Integration (reuse) opportunities are identified using register names only, contrary to [1, 32], therefore reuse is identified in the frontend. Yet, register sharing is once again achieved through per-register counters.

8. CONCLUSION

Move elimination (ME) is a currently implemented optimization that mitigates the high number of register-to-register moves in x86. Speculative Memory Bypassing (SMB), while not implemented to our knowledge, is an attractive solution to mitigate the cost of communicating results through the memory hierarchy rather than registers, although it has been shown to be of limited interest in the presence of a good memory dependency predictor [4].

Both optimizations require the ability for physical registers to be shared by several instructions, yet previously proposed mechanisms were either too limited (e.g., MIT [12]), or too costly/unadapted in case of pipeline flushes [22].

In this work, we first proposed an implementation of SMB that identifies both load-load and store-load pairs through a TAGE-like Instruction Distance Predictor. Using this distance, a ROB index is generated and the physical register identifier to be used by the bypassing instruction is retrieved. We also pointed out that since the ROB is used to bypass, recently committed instructions are still reachable for SMB. However, on our benchmark set, bypassing from recently committed instructions did not improve performance noticeably and was in fact often detrimental.

Second, and more importantly, we proposed a register sharing mechanism that can be adapted to any optimization requiring sharing, and that lends itself to checkpointing. That is, correct state can be recovered on a pipeline flush by restoring checkpointed state and simple comparisons on narrow values. Hence, recovery can be done in a single cycle.

Finally, we considered using a small fully-associative buffer, the ISRB, implementing our tracking mechanism for a subset of the physical registers. We showed that only 32 entries of two 3-bit fields are required to obtain most of the potential of both ME and SMB. In particular, average speedup is 5.5% and up to 39.6%. We point out that considering only one of both mechanisms puts less pressure on the ISRB, hence less entries need to be implemented (respectively 16/8 for ME and 24/16 for SMB).

Acknowledgments

This work was partially supported by the European Research Council Advanced Grant DAL No. 267175

9. REFERENCES

[1] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," in *Proceedings of the International Symposium on Microarchitecture*, pp. 216–225, Nov 1998.

[2] Intel Corp., "Software optimization manual," September 2014.

[3] T. Sha, M. M. K. Martin, and A. Roth, "NoSQ: Store-load communication without a store queue," in *Proceedings of the International Symposium on Microarchitecture*, pp. 285–296, IEEE Computer Society, 2006.

[4] G. H. Loh, R. Sami, and D. H. Friendly, "Memory bypassing: Not worth the effort," in *Proceedings of the 1st Workshop on Duplicating, Deconstructing, and Debunking*, pp. 71–80, 2002.

[5] G. S. Tyson and T. M. Austin, "Memory renaming: Fast, early and accurate processing of memory communication," *International Journal of Parallel Programming*, vol. 27, pp. 357–380, Oct. 1999.

[6] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 235–245, Dec. 1997.

[7] A. Moshovos and G. S. Sohi, "Read-after-read memory dependence prediction," in *Proceedings of the International Symposium on Microarchitecture*, pp. 177–185, 1999.

[8] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *Proceedings of the International Symposium on Computer Architecture*, pp. 25–34, 2002.

[9] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, pp. 1–23, 2006.

[10] Advanced Micro Device, "Software optimization guide for amd family 10h and 12h processors," February 2011 2011.

[11] Intel Corp., "Intel 64 and ia-32 architectures software developers manual," June 2015.

[12] S. Raikin, D. J. Sager, Z. Sperber, E. Krimer, O. Lempel, S. Shwartsman, A. Yoaz, and O. Golz, "Tracking mechanism coupled to retirement in reorder buffer for indicating sharing logical registers of physical register in record indexed by logical register," Dec. 16 2014. US Patent 8,914,617.

[13] S. Önder and R. Gupta, "Load and store reuse using register file contents," in *Proceedings of the International Conference on Supercomputing*, pp. 289–302, 2001.

[14] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers, Table 16.2," Aug. 2014.

[15] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 198–, 2004.

[16] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 142–153, 1998.

[17] M. Golden, S. Arekapudi, and J. Vinh, "40-entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86_64 core," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 80–82, 2011.

[18] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint processing and recovery: towards scalable large instruction window processors," in *Proceedings of the International Symposium on Microarchitecture*, pp. 423–434, Dec 2003.

[19] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2014.

[20] D. Leibholz and R. Razdan, "The Alpha 21264: a 500 MHz out-of-order execution microprocessor," in *Proceedings of IEEE Compcon'*, pp. 28–36, 1997.

[21] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, pp. 28–40, Apr. 1996.

[22] A. Roth, "Physical register reference counting," *Computer Architecture Letters*, vol. 7, pp. 9–12, Jan 2008.

[23] S. Battle, A. Hilton, M. Hempstead, and A. Roth, "Flexible register management using reference counting," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 1–12, 2012.

[24] S. Sundar and C. Blasco-Allue, "RDA checkpoint optimization," Feb. 5 2015. US Patent App. 13/955,847.

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[26] T. Nowatzki, J. Menon, C. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *Micro, IEEE*, vol. PP, no. 99, pp. 1–1, 2015.

[27] Standard Performance Evaluation Corporation, "CPU2000."

[28] Standard Performance Evaluation Corporation, "CPU2006."

[29] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 244–, 2003.

[30] V. Petric, A. Bracy, and A. Roth, "Three extensions to register integration," in *Proceedings of the International Symposium on Microarchitecture*, pp. 37–47, 2002.

[31] J. Yang and R. Gupta, "Energy-efficient load and store reuse," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 72–75, 2001.

[32] A. Sodani and G. Sohi, "Dynamic instruction reuse," in *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pp. 194–205, June 1997.

[33] A. Gonzalez, J. Gonzalez, and M. Valero, "Virtual-physical registers," in *Proceedings of the Symposium on High-Performance Computer Architecture*, pp. 175–184, 1998.

[34] V. Petric, T. Sha, and A. Roth, "Reno: a rename-based instruction optimizer," in *Proceedings of the International Symposium on Computer Architecture*, pp. 98–109, June 2005.

[35] B. Fahs, T. Rafacz, S. Patel, and S. S. Lumetta, "Continuous optimization," in *Proceedings of the International Symposium on Computer Architecture*, pp. 86–97, June 2005.